CHAPTER 17

PICOBLAZE INTERRUPT INTERFACE

17.1 INTRODUCTION

During normal program execution, a microcontroller *polls* the I/O peripherals (i.e., checks the status signals) and determines the course of action accordingly. An I/O peripheral is passive and waits for its turn. The *interrupt* is a mechanism that allows an external I/O peripheral to initiate the operation. It, as the name shows, interrupts normal program execution and starts a service routine for the I/O peripheral. For a microcontroller, the interrupt is usually reserved for a time-critical peripheral operation, which must be processed immediately. The PicoBlaze microcontroller provides support for simple interrupt-handling capability. In this chapter, we examine the PicoBlaze's interrupt mechanism and use an example to illustrate software and interface development.

17.2 INTERRUPT HANDLING IN PICOBLAZE

Interrupt handling is a coordinated effort between hardware and software. When an external peripheral needs service through interrupt, it asserts the `interrupt` signal of PicoBlaze. If the interrupt service is enabled, PicoBlaze completes execution of the current instruction, activates the `interrupt_ack` signal to acknowledge the acceptance of the interrupt request, and then implicitly executes the **call** 3FF instruction. When the instruction is executed, the current content of the program counter is saved in stack and the 3FF address is loaded to the programmer counter. Note that the 3FF address is the last location in the instruction

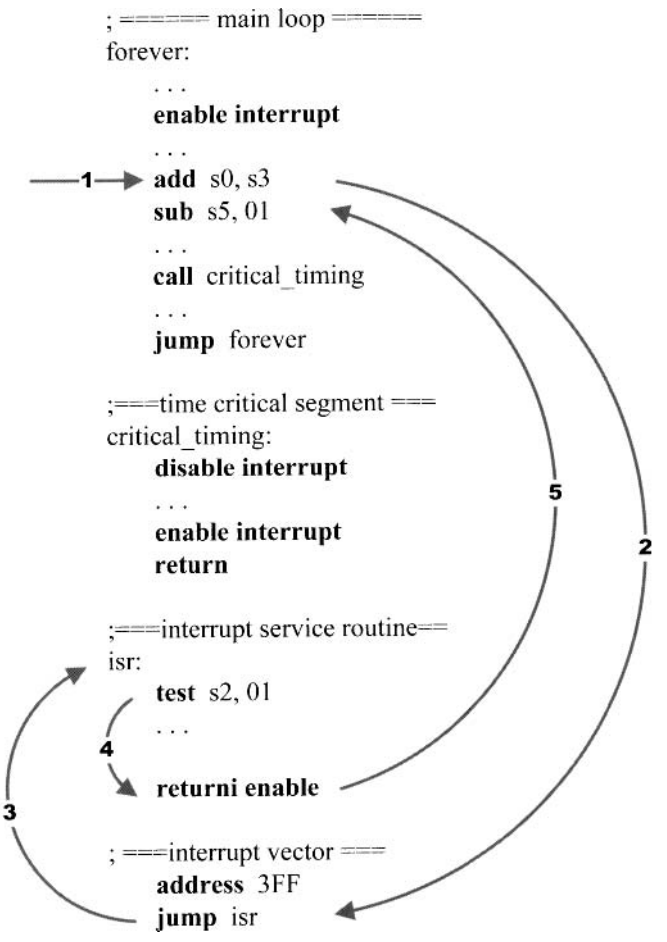

Figure 17.1 Interrupted flow.

memory and serves as the starting point of the interrupt service routine. It usually contains a **jump** instruction, which leads to the body of the service routine. The service should be ended with a **returni** instruction to return to the interrupted point and resume the previous execution.

17.2.1 Software processing

Four instructions are associated with interrupt, as discussed in Section 14.5.9. The **enable interrupt** and **disable interrupt** instructions enable and disable the interrupt request, and the two return-from-interrupt instructions, **returni enable** and **returni disable**, return execution to the interrupted point.

A typical program segment with interrupt service routine is shown in Figure 17.1. It generally consists of the following segments:

- *An initial* **enable interrupt** *instruction*: used to enable the interrupt service. This is needed since the interrupt request is disabled by default.

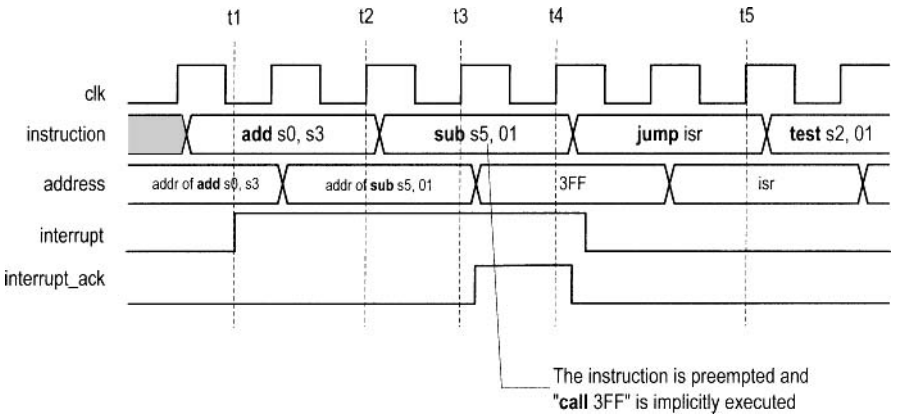

Figure 17.2 Timing diagram of an interrupt event.

- *A* **jump** *instruction in the end of the instruction memory* (i.e., 3FF): leads to the interrupt service routine.
- *Interrupt service routine*: the code that actually performs the requested service. The routine should be ended with a **returni** instruction.

A representative flow of an interrupt event is shown in Figure 17.1. We assume that the external I/O assert the `interrupt` signal in the middle of the **add** `s0,s3` instruction. PicoBlaze performs the following steps in sequence:

1. Completes execution of the current execution.
2. Saves the content of the program counter, clears the interrupt flag, `i`, to zero, preserves the zero and carry flags, and loads the program counter with 3FF.
3. Executes the **jump** `isr` instruction in the 3FF address.
4. Performs the service routine.
5. Executes the **returni** instruction, in which the saved program counter and flags are restored.
6. Resumes the interrupted program and executes the **sub** `s5,01` instruction.

17.2.2 Timing

The detailed timing diagram of the previous interrupt event is shown in Figure 17.2. The basic sequence is:

- At `t1`: The external interrupt interface asserts the `interrupt` signal. PicoBlaze continues the normal operation to complete execution of the current **add** `s0,s3` instruction.
- At `t2`: PicoBlaze recognizes the interrupt and aborts the next instruction (**sub** `s5,01`) and implicitly executes the **call** 3FF instruction.
- At `t3`: PicoBlaze asserts the `interrupt_ack` signal. It also saves the address of the **sub** `s5,01` instruction, preserves the zero and carry flags, and clears the interrupt flag to 0.
- At `t4`: PicoBlaze loads and executes the instruction in address 3FF, **jump** `isr`. The external interrupt interface circuit acknowledges the `interrupt_ack` signal and deasserts the `interrupt` signal.

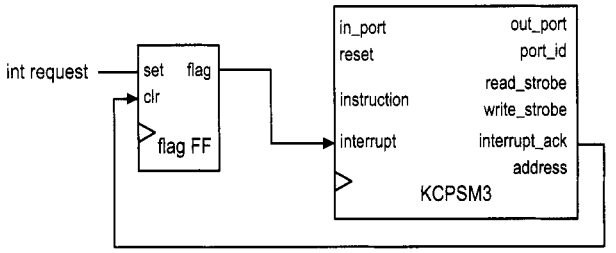

Figure 17.3 Interrupt interface with a single request.

- At `t5`: PicoBlaze starts the interrupt service routine.

Note that it requires up to five clock cycles from the time that the `interrupt` signal is asserted to the time that the first instruction of interrupt service routine is executed.

17.3 EXTERNAL INTERFACE

The nature of the interrupt request is similar to that of a single-access port discussed in Section 16.3.2. After the request is accepted, it must be cleared so that the same request will not be processed multiple times. The flag FF discussed in Section 7.2.4 can be used for this purpose.

17.3.1 Single interrupt request

If there is only one I/O peripheral in a PicoBlaze system that can generate an interrupt request, we just need a single flag FF in the interrupt interface circuit, as shown in Figure 17.3. When the service is required, the external I/O circuit asserted the `int request` signal for one clock cycle, which sets the flag FF to '1' and activates the `interrupt` input of PicoBlaze. If the interrupt is enabled in PicoBlaze, it acknowledges acceptance of the request by asserting the `interrupt_ack` signal for one clock cycle, which clears the flag FF to '0'.

17.3.2 Multiple interrupt requests

Processing a PicoBlaze system with two or more interrupt requests is more involved. The PicoBlaze microcontroller must determine which peripheral issues the request and clear the corresponding flag FF after the request is accepted. This needs the coordination of the hardware interface and the interrupt service routine.

The interrupt interface with two requests is shown in Figure 17.4. The two individual requests, `int request0` and `int request1`, are connected to two flag FFs, and the output signals of the FFs are passed to an or gate to generate the final interrupt request signal. In addition, the two signals are also routed to the input multiplexer. If at least one request is asserted, the `interrupt` signal of PicoBlaze is asserted. When PicoBlaze senses the request, it does not know which peripheral or whether both peripherals issue the request. The interrupt service routine must first input the two request signals and check their values according to the assigned priority, and then perform the corresponding service.

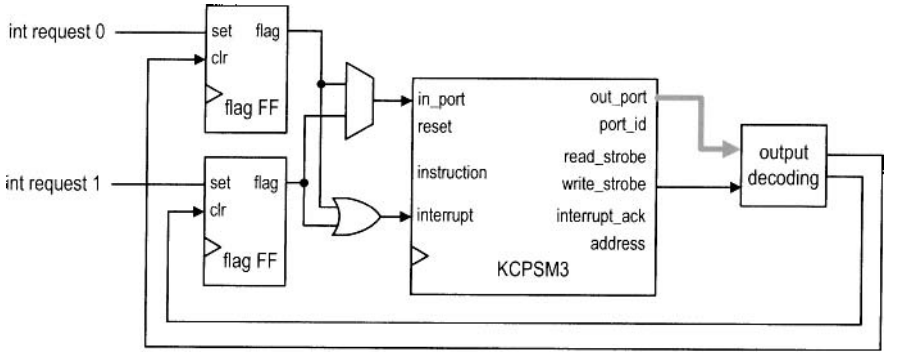

Figure 17.4 Interrupt interface with two requests.

In addition, PicoBlaze also needs to clear the corresponding flag FF. The `interrupt_ack` signal cannot be used for this purpose because it is not known which peripheral's request is accepted when the `interrupt_ack` signal asserted. Instead, we need to use a special output decoding circuit to generate a clear tick. The `clr` signal of each flag FF is assigned to a unique port id. In the interrupt service routine, we add an **output** instruction after determining which interrupt request is accepted. The instruction does not actually output any data. It is used to generate a single-clock-cycle tick to clear the corresponding flag FF.

To reduce the software overhead and increase response speed, we can design an *interrupt controller* to facilitate the process. This approach is discussed in Experiment 17.7.5.

17.4 SOFTWARE DEVELOPMENT CONSIDERATIONS

17.4.1 Interrupt as an alternative scheduling scheme

Recall that a microcontroller-based application usually follows a simple polling program structure:

```
        call initialization_routine
forever:
        call task1_routine;
        call task2_routine;
         ...
        call taskn_routine;
        jump forever;
```

Some tasks may involve I/O operations. During execution, the microcontroller checks the I/O status in turn and takes actions accordingly. The program structure implicitly implements a *round-robin* schedule, in which each task waits in turn to be executed. This scheme can work properly if the loop interval is short enough so that each I/O request can be checked and processed in a timely manner. In some applications, there may exist one or two time-critical I/O requests that require immediate attention. The interrupt mechanism provides a way to alter the original schedule and gives certain tasks higher priorities.

Since an interrupt can occur at any time, the original loop must consider the frequency of interrupt and the required service time of each interrupt request. This can be complicated if there are multiple interrupt requests and the service routine is involved.

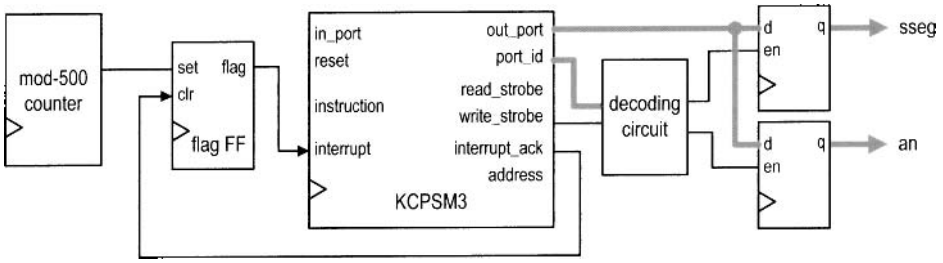

Figure 17.5 Interrupt interface with a timer.

17.4.2 Development of an interrupt service routine

The interrupt service routine is somewhat like a subroutine. It suspends normal program execution, performs an independent task, and then resumes the previous execution. However, unlike a subroutine call, an interrupt can occur any time. To resume execution later, the service routine must save the *current state* (also known as the *context*) of the PicoBlaze processor. In other words, the service routine must save all registers used in service routine computation and then restore them before returning to normal execution. This process is known as *context switching*.

Since PicoBlaze is a compact 8-bit microcontroller, the hardware support for context switching and scheduling is very limited. We should use the polling scheme in general and keep the interrupt structure simple and straightforward. Instead of worrying about context switching, we can allocate several dedicated registers to be used exclusively in the interrupt service routine.

17.5 DESIGN EXAMPLE

The square circuit of Chapter 16 uses a seven-segment LED display to show the values of input operands and result. We use the predesigned LED multiplexing module, `disp_mux`, for this purpose. The design of this module is discussed in Section 4.5.1. It consists of a large counter to generate slow enable pulses and a multiplexing circuit to route the input patterns.

To save hardware, we can implement this functionality in software and let PicoBlaze control the 4-bit enable signal, `an`, and the 8-bit LED signal, `sseg`, of the four-digit LED display directly. To generate a visually continuous pattern, the enable pulse and LED patterns must be refreshed at a constant rate, as shown in Figure 4.6. While using pure software to keep track of time is possible, the code is tedious and error-prone. We use a dedicated hardware timer and PicoBlaze's interrupt facility to perform the task. The required hardware and software modifications are illustrated in the following subsections.

17.5.1 Interrupt interface

The block diagram of the timer and interrupt interface, as well as the new output buffers, is shown in Figure 17.5. The timer is a mod-500 counter and generates a single-clock-cycle tick every 500 clock cycles. Since the 50-MHz clock is used for the timer, the period of the tick is 0.01 ms. Because there is only one interrupt request, we use the flag FF scheme

discussed in Section 17.3.1 for the interrupt interface. The tick sets the flag FF and activates the `interrupt` signal of PicoBlaze.

17.5.2 Interrupt service routine development

To keep track of the elapsed time, PicoBlaze counts the number of timer ticks. As discussed in Section 17.4.2, we want to keep the interrupt service routine simple and use two dedicated registers, `count_msb` and `count_lsb`, for this task. The two registers are cascaded as a 16-bit register and are incremented each time the interrupt service routine is called. They can count to 0.6 second (i.e., $2^{16} * 0.01$ ms). The interrupt-related code segment is

```
    namereg se, count_msb   ;timer tick count 8 MSBs
    namereg sf, count_lsb   ;timer tick count 8 LSBs
    ...
;interrupt service routine
int_service_routine:
    add count_lsb, 01       ;inc 16-bit counter
    addcy count_msb, 00
    returni enable

;interrupt vector
    address 3FF
    jump int_service_routine
```

17.5.3 Assembly code development

With the timing information available, we can derive a new subroutine, `display_mux_out`, for the LED display. This routine replaces the `disp_led` routine used in Chapter 16. Two new output buffers are needed to store the an and sseg signals, as shown in Figure 17.5. The main task of the subroutine is to store the an pattern, which can be "1110", "1101", "1011", or "0111", and the corresponding seven-segment LED pattern to the registers periodically. As discussed in Section 4.5.1, the refreshing rate should be around from a few hundred to a few thousand hertz. In our code we update these registers every 2^{10} ticks, which is about 10 ms. We also use a register, `led_pos`, to keep track of the current display position (i.e., one of the four LED displays).

To incorporate the new interrupt feature into Listing 16.3, the code is modified as follows:

- Add new port and register definitions.
- Replace the original `disp_led` routine with the `display_mux_out` routine.
- Add the **enable interrupt** instruction in the `init` routine to enable interrupt handling.
- Initialize the `led_pos`, `count_msb`, and `count_lsb` registers in the `init` routine.
- Add the interrupt service routine.

The modified portion of the assembly code is shown in Listing 17.1.

Listing 17.1 Square program with interrupt interface

```
...
;register alias
namereg sb, led_pos     ;led disp position (0, 1, 2 or 3)
namereg se, count_msb   ;timer tick count 8 MSBs
namereg sf, count_lsb   ;timer tick count 8 LSBs
...
```

```
;output port definitions
constant an_port, 00
constant sseg_port, 01
...
; main program
   call init                ;initialization
forever:
   ;main loop body
   call proc_btn            ;check & process buttons
   call square              ;calculate square
   call load_led_pttn       ;store led patterns to ram
   call display_mux_out     ; multiplex led patterns
   jump forever

;=============================================================
;routine: init
;=============================================================
init:
   enable interrupt
   ...
   load led_pos, 00
   load count_msb, 00
   load count_lsb, 00
   return

;=============================================================
;routine: display_mux_out
;  function: generate enable pulse & led pattern
;            for 4-digit 7-segment led display
;  input register:
;     count_msb, count_lsb: timer count
;     led_pos: current led position
;  output register:
;     led_pos: updated led position
;  tmp register: data, addr
;=============================================================
display_mux_out:
   compare count_msb, 02   ;count=00000100_00000000
   jump c, mux_out_done
   ;clear time counter (count > 20)
   load count_lsb, 00
   load count_msb, 00
   ;update 7-segment led position
   add led_pos, 01
   compare led_pos, 04
   jump nz, gen_an_signal
   load led_pos, 00        ;led_pos wraps around
gen_an_signal:
   ;generate 4-bit anode enable signal
   load data, 0E           ;xxxx_1110
   compare led_pos, 00
   jump z, shift_an_0
   compare led_pos, 01
```

```
    jump z, shift_an_1
    compare led_pos, 02
    jump z, shift_an_2
    sl1 data                    ;shift 1110 3 times
shift_an_2:
    sl1 data                    ;shift 1110 2 times
shift_an_1:
    sl1 data                    ;shift 1110 1 times
shift_an_0:
    output data, an_port
    ;output 7-seg led pattern
    load addr, led0
    add addr, led_pos
    fetch data, (addr)
    output data, sseg_port
mux_out_done:
    return

;=============================================================
;routine: interrupt service routine
;  function: increment 16-bit counter
;  input register:
;     count_msb, count_lsb: timer count
;  output register:
;     count_msb, count_lsb: incremented
;=============================================================
int_service_routine:
    add count_lsb, 01       ;inc 16-bit counter
    addcy count_msb, 00
    returni enable

;=============================================================
;interrupt vector
;=============================================================
    address 3FF
    jump int_service_routine

;=============================================================
;The following are the same as the previous Listing:
;  proc_btn, load_led_pttn,
;  hex_to_led, get_lower_nibble, get_upper_nibble
;  square, mult_soft
;  ...
;=============================================================
```

17.5.4 VHDL code development

The I/O interface of the interrupt-based square circuit includes three parts. The input interface is similar to that in Section 16.4. The output interface consists of a decoding circuit and two output registers for the an and sseg signals, as shown on the right of Figure 17.5. The interrupt interface consists of a timer and a flag FF, as shown on the

left of Figure 17.5. The HDL code basically follows the block diagram and is shown in Listing 17.2.

Listing 17.2 PicoBlaze-based square circuit with interrupt

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pico_int is
   port(
      clk, reset: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(1 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end pico_int;

architecture arch of pico_int is
   -- KCPSM3/ROM signals
   signal address: std_logic_vector(9 downto 0);
   signal instruction: std_logic_vector(17 downto 0);
   signal port_id: std_logic_vector(7 downto 0);
   signal in_port, out_port: std_logic_vector(7 downto 0);
   signal write_strobe, read_strobe: std_logic;
   signal interrupt, interrupt_ack: std_logic;
   -- I/O port signals
   -- output enable
   signal en_d: std_logic_vector(1 downto 0);
   -- four-digit seven-segment led display
   signal sseg_reg: std_logic_vector(7 downto 0);
   signal an_reg: std_logic_vector(3 downto 0);
   -- two pushbuttons
   signal btnc_flag_reg, btnc_flag_next: std_logic;
   signal btns_flag_reg, btns_flag_next: std_logic;
   signal set_btnc_flag, set_btns_flag: std_logic;
   signal clr_btn_flag: std_logic;
   -- interrupt-related signals
   signal timer_reg, timer_next: unsigned(8 downto 0);
   signal ten_us_tick: std_logic;
   signal timer_flag_reg, timer_flag_next: std_logic;
begin
   --=====================================================
   -- I/O modules
   --=====================================================
   btnc_db_unit: entity work.debounce
      port map(
         clk=>clk, reset=>reset, sw=>btn(0),
         db_level=>open, db_tick=>set_btnc_flag);
   btns_db_unit: entity work.debounce
      port map(
         clk=>clk, reset=>reset, sw=>btn(1),
         db_level=>open, db_tick=>set_btns_flag);
   --=====================================================
```

```
50  -- KCPSM and ROM instantiation
    --==================================================
    proc_unit: entity work.kcpsm3
       port map(
          clk=>clk, reset =>reset,
55        address=>address, instruction=>instruction,
          port_id=>port_id, write_strobe=>write_strobe,
          out_port=>out_port, read_strobe=>read_strobe,
          in_port=>in_port, interrupt=>interrupt,
          interrupt_ack=>interrupt_ack);
60  rom_unit: entity work.int_rom
       port map(
           clk => clk, address=>address,
           instruction=>instruction);
    --==================================================
65  -- output interface
    --==================================================
    --     outport port id:
    --       0x00: an
    --       0x01: ssg
70  --==================================================
    -- registers
    process (clk)
    begin
       if (clk'event and clk='1') then
75        if en_d(0)='1' then
             an_reg <= out_port(3 downto 0);
          end if;
          if en_d(1)='1' then sseg_reg <= out_port; end if;
       end if;
80  end process;
    an <= an_reg;
    sseg <= sseg_reg;
    -- decoding circuit for enable signals
    process(port_id,write_strobe)
85  begin
       en_d <= (others=>'0');
       if write_strobe='1' then
          case port_id(0) is
             when '0' => en_d <="01";
90           when others => en_d <="10";
          end case;
       end if;
    end process;
    --==================================================
95  -- input interface
    --==================================================
    --     input port id
    --       0x00: flag
    --       0x01: switch
100 --==================================================
    -- input register (for flags)
    process(clk)
```

```
   begin
      if (clk'event and clk='1') then
         btnc_flag_reg <= btnc_flag_next;
         btns_flag_reg <= btns_flag_next;
      end if;
   end process;

   btnc_flag_next <= '1' when set_btnc_flag='1' else
                     '0' when clr_btn_flag='1' else
                      btnc_flag_reg;
   btns_flag_next <= '1' when set_btns_flag='1' else
                     '0' when clr_btn_flag='1' else
                      btns_flag_reg;
   -- decoding circuit for clear signals
   clr_btn_flag <='1' when read_strobe='1' and
                           port_id(0)='0' else
                  '0';
   -- input multiplexing
   process(port_id,btns_flag_reg,btnc_flag_reg,sw)
   begin
      case port_id(0) is
         when '0' =>
            in_port <= "000000" &
                       btns_flag_reg & btnc_flag_reg;
         when others =>
            in_port <= sw;
      end case;
   end process;
   -- ======================================================
   --  interrupt interface
   -- ======================================================
   -- 10 us counter
   process(clk)
   begin
      if (clk'event and clk='1') then
         timer_reg <= timer_next;
      end if;
   end process;
   timer_next <= (others=>'0') when timer_reg=499 else
                 timer_reg+1;
   ten_us_tick <= '1' when timer_reg=499 else '0';
   -- 10 us tick flag
   process(clk)
   begin
      if (clk'event and clk='1') then
         timer_flag_reg <= timer_flag_next;
      end if;
   end process;
   timer_flag_next <= '1' when ten_us_tick='1' else
                      '0' when interrupt_ack='1' else
                      timer_flag_reg;
   -- interrupt request
   interrupt <= timer_flag_reg;
```

```
end arch;
```

17.6 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapters 14 to 16.

17.7 SUGGESTED EXPERIMENTS

17.7.1 Alternative timer interrupt service routine

The interrupt service routine in Listing 17.1 uses two dedicated registers to record the number of timer ticks. The two registers thus cannot be used for other computation. An alternative is to use 2 bytes of the data RAM for this purpose and use the registers only temporarily in the service routine. Since interrupt can occur anytime, we must save and restore the corresponding registers. For example, if the `s0` and `s1` registers are used in the service routine for computation, their contents must be saved when the service routine is invoked and then restored later when the computation is completed. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.2 Programmable timer

We can replace the mod-500 counter of Section 17.5 with a general mod-m counter and thus make the timer "programmable." The new timer operates as follows:

- m is a 12-bit unsigned number.
- The four LSBs of m is "1111".
- The timer has an 8-bit register to store the eight MSBs of m. The register is treated as a new output port of PicoBlaze.
- A new pushbutton controls the loading of the register. When it is pressed, PicoBlaze inputs the value from the 8-bit switch and outputs the value to the timer's register.

Design the new I/O interface, derive the assembly and HDL codes, and compile and synthesize the circuit. Load different values in the timer and observe what happens to the LED display.

17.7.3 Set-button interrupt service routine

In the square circuit discussed in Section 16.4, the `s` button is used to load the a and b operands from the 8-bit switch. Its status is polled continuously in the main loop. We can revise this portion of the code and use an interrupt mechanism to perform this task. The interrupt service routine involves several temporary registers, and they must be saved and restored properly, as discussed in Experiment 17.7.1. Design the new I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.4 Interrupt interface with two requests

Assume that we want to implement both the timer interrupt request of Listing 17.1 and the set-button interrupt request of Experiment 17.7.3 in a PicoBlaze system. Follow the

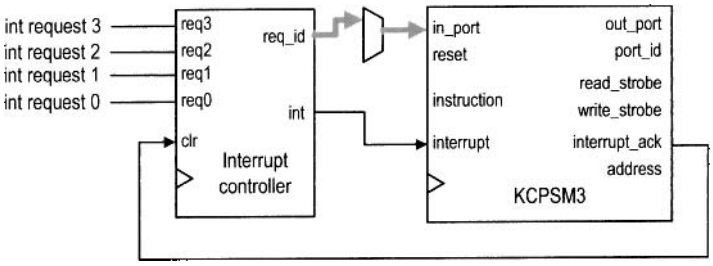

Figure 17.6 Interrupt interface with a four-request interrupt handler.

discussion in Section 17.3.2 to design the new interrupt interface and interrupt service routine. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.5 Four-request interrupt controller

An interrupt controller helps the processor to process multiple interrupt requests. The block diagram of a four-request interrupt controller is shown in Figure 17.6. The interrupt controller should contain four flag FFs and a special priority encoding circuit. If one or more interrupt requests are activated, the controller determines which request has the highest priority, places its 2-bit code on the `req_id` port, and asserts the `int` signal. When PicoBlaze asserts the `interrupt_ack` signal, the controller clears the corresponding flag. For simplicity, we assume that `int_request_3` has the highest priority and `int_request_0` has the lowest priority.

Derive HDL code for the interrupt controller and repeat Experiment 17.7.4 using the new controller (the two unused interrupt requests can be tied to '0').